VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

System pro sestavování a automatické nasazování projektů

System for Building and Automating Deploying Projects

2018 Bc. Lukáš Drábek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Bc. Lukáš Drábek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: System pro sestavování a automatické nasazování projektů
System for Building and Automating Deploying Projects

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem bakalářské práce je získat přehled v oblasti Průběžné integrace (Continuous Integration). Student vytvoří State of the Art v dané oblasti, následně provede instalaci a konfigurace vybraného software. Výstupem bude nasazení aplikace přes Průběžnou integrace.

1. Nastudovat problematiku průběžné integrace, její výhody a případně i nevýhody.
2. Dle pokynů vedoucího nainstalovat a nakonfigurovat vybraný software.
3. Otestovat funkčnost s propojením na úložiště GIT a SVN.

Seznam doporučené odborné literatury:

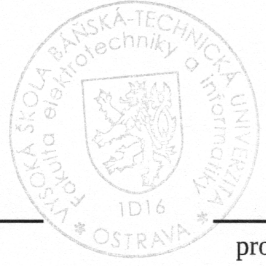
[1] Paul M. Duvall, Steve Matyas, Andrew Glover. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional; 1 edition (July 9, 2007), ISBN-10: 9780321336385, ISBN-13: 978-0321336385
[2] Jez Humble , David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler)). Addison-Wesley Professional; 1 edition (August 6, 2010). ISBN-10: 0321601912, ISBN-13: 978-0321601919
[3] Kent Beck. Test Driven Development: By Example. Addison-Wesley Professional; 1 edition (November 18, 2002). ISBN-10: 0321146530, ISBN-13: 978-0321146533

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Lukáš Vojáček, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018



doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal, a nejsou mi známy žádné okolnosti, které by mohly vést k pochybnostem o mojí práci.

Lukáš Drábek

Datum odevzdání bakalářské práce: 27.4.2018

Poděkování

Rád bych na tomto místě poděkoval především vedoucímu mé bakalářské práce, panu Ing. Lukáši Vojáčkovi, Ph.D. za vedení, cenné rady a trpělivost.

Abstrakt

Tato bakalářská práce poskytuje obecný náhled na problematiku průběžné integrace. Dále poskytuje demonstraci použití softwaru pro realizaci průběžné integrace.

Po prozkoumání možností softwaru v oblasti průběžné integrace byl zvolen opensource projekt Jenkins, který byl následně nainstalován a nakonfigurován na poskytnutý server.

K ověření funkčnosti došlo k vytvoření triviálního projektu umístěného na uložiště GIT a SVN. Následně došlo k vytvoření úloh v softwaru Jenkins. Následné ověřování funkčnosti průběžné integrace s oběma uložišti proběhlo úspěšně.

Klíčová slova: průběžná integrace, jenkins, extrémní programování, LaTeX, bakalářská práce

Abstract

This bachelor thesis provides a general overview of the issues of continuous integration. It also provides a demonstration of using software to implement continuous integration.

After examining the possibilities of continuous integration software, the Jenkins opensource project was selected, which was subsequently installed and configured on the provided server.

To verify functionality, a trivial project was created on the GIT and SVN storage. Subsequently, tasks in Jenkins were created. Subsequent verification of the functionality of continuous integration with both GIT and SVN was successful.

Key Words: continuous integration, jenkins, extreme programming, LaTeX, bachelor thesis

Obsah

Seznam použitých zkratek a symbolů

.NET	–	Zastřešující název pro soubor technologií v softwarových produktech
access token	–	Přístupová značka sloužící k identifikaci
Android	–	Mobilní operační systém vyvinutý společností Google
CI	–	Continuous integration - průběžná integrace
CRUD	–	Zkratka zastřešující čtyři základní operace - vytvoření, čtení, editaci a smazání
CVS	–	Concurrent Version System - systém pro správu verzí
Framework	–	Softwarova struktura pro podporu programování
GUI	–	Grafické uživatelské rozhraní
Hardware	–	Technické vybavení
iOS	–	Mobilní operační systém od společnosti Apple
Jira	–	Software pro organizování práce
LDAP	–	Protokol pro ukládání a přístup k datům na adresářovém serveru
PaaS	–	Platforma jako služba
PowerShell	–	Skriptovací jazyk a shell od Microsoftu
RCS	–	Software pro správu verzí
SHA-1	–	Rozšířená hashovací funkce
Software	–	Programové vybavení
TRX	–	Soubor s výsledky testů v XML formátu
VPN	–	Virtuální privátní síť
WebDav	–	Protokol poskytující možnost vzdálené správy souborů na webovém serveru
WWW	–	Světová komunikační síť
Xamarin	–	Nástroj pro vývoj aplikací pro různé platformy
XML	–	Rozšiřitelný značkovací jazyk používaný pro snadnou výměnu dat

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Při vývoji softwaru prochází programátor několika stádii. V počátku jen píše kód a doufá, že bude fungovat. S postupným nabýváním zkušeností se dostává z fáze "ono to skutečně něco dělá"do stavu, kdy může říci: "Skutečně to dělá, co chci". Netrvá dlouho a dostaví se uvědomění jakou sílu může mít programovací jazyk a první větší projekt. Nicméně už nelze efektivně psát kód vrstvením funkcí a proměnných. Zde je třeba dát programování řád.

Tohoto řádu lze dosáhnout použitím různých vývojových praktik a nástrojů. Jednou z těchto praktik je i průběžná integrace. Jde o metodiku vývoje softwaru odvozenou z extrémního programování.

V této práci budou uvedeny jednotlivé praktiky, ze kterých průběžná integrace vychází. Dále bude uvedeno, jaké nástroje jsou kritické pro průběžné programování doplněné o přehled aktuálních softwarových řešeních na trhu. Na závěr bude uveden demonstrativní příklad použití softwaru pro průběžnou integraci.

1.1 Stručný Souhrn kapitol

V následující kapitole [2] bude shrnuto co průběžná integrace obnáší. V podkapitolách bude popsáno extrémní programování [2.5]. Dále nahlédneme na správu verzí zdrojových kódů [2.6] a lehce nahlédneme na téma testování [2.7]. V závěru kapitoly bude rozebráno co je třeba automatizovat pro funkci průběžné integrace [2.8]

Další kapitola [3] podá letmý přehled aktuálních software určených pro průběžnou integraci.

V závěru práce [4] uvedu práci s vybraným softwarem. Základní informace o něm doplněné o ukázku nastavení a vytvoření úlohy průběžné integrace.

2 Průběžná integrace

Průběžná integrace je souhrn různých vývojářských nástrojů a praktik v softwarovém inženýrství sloužící k udržovaní „zdravého“ kódu při týmovém vývoji. Jednotliví vývojáři integrují svůj kód alespoň jednou denně. To má za následek několik integrací kódu za den. Každou integrace kódu pak doprovází automatické testování sloužící k rychlé detekci chyb.

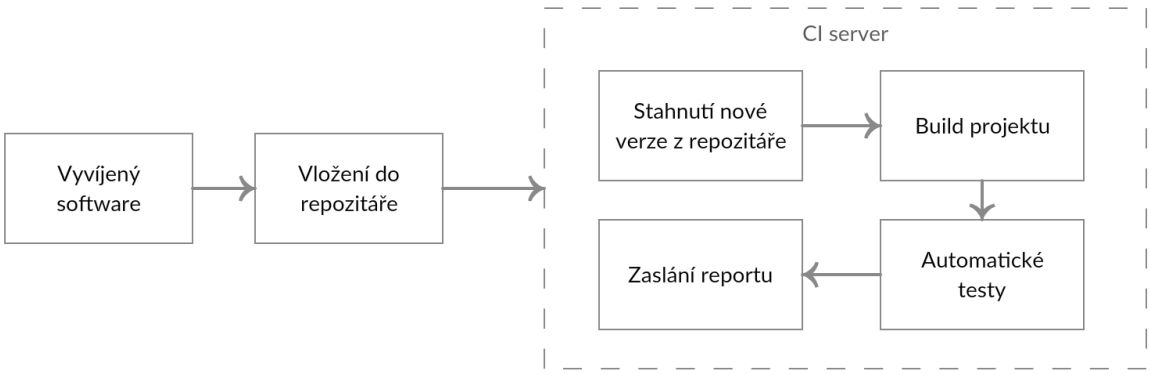
Průběžná integrace pochází z paradigmatu extrémního programování, ale jeho principy lze aplikovat na jakýkoliv iterativní model programování. Příkladem iterativního modelu je programování řízené testy. Hlavni naplní průběžné integrace je častá integrace kódu, automatické sestavování kódu a následné spouštění testů kódu. To má za následek urychlení vývoje softwaru.

Pro nasazeni průběžné integrace existuje několik důvodu. Mimo již zmíněného urychlení vývoje jde především o snížení rizika výskytu chyb v softwaru. Protože typicky procesy průběžné integrace obsluhuje softwarový správce běžící na serveru, dochází k redukci problémů spojených se scénářem „na mem stroji to pracovalo“. Důvěra v kód je dalším z důvodů. Jsou-li vytvořené robustní testovací scénáře, získáváme po jejich úspěšném provedeni jistotu, že nezavadíme do společného projektu chybu.

Pro úspěšné fungování průběžné integrace se neobejdeme bez několika nutných nástrojů. Z principu funkcionality se neobejdeme bez centrálního uložiště kódu. Typicky jde o nástroje typu Apache Subversion, nebo Git. Dalšími předpoklady jsou nástroje obstarávající automatizaci sestavení a testování. Užitečným doplňkem jsou nástroje pro kontrolu kvality kódu.

Nutno podotknout, že nasazením průběžné integrace nedojde k redukci chyb jako takových, ale jejich snazší a rychlejší odhalení. To má za následek snížení dopadu chyb ve vývoji softwaru. Z tohoto důvodu by měl každý vývojář svou změnu v kódu integrovat často a pravidelně.

Jak již bylo řečeno všechny procesy obstarává specializovaný software běžící na serveru. Díky tomu lze relativně jednoduše zapojit do procesu vývoje kódu i další služby. Příkladem může být komunikace skrze nástroje podporující komunikaci v tymu – Slack, Discord apod. Celý proces může být také doplněn o automatické nasazovaní softwaru.[3, 2, 8]



Obrázek 1: Model průběžné integrace

2.1 Výhody průběžné integrace

Mezi hlavní výhody zapojení této metodiky je redukce rizika. Tím, že během dne dochází často k integraci kódu, nám průběžná integrace poskytuje jakousi záchrannou síť, snižující riziko zavedení chyby hlouběji do projektu. Každou integraci doprovází automatické testy, díky kterým dochází k odhalení chyb a jejich opravě mnohem rychleji. Zavedením průběžné integrace můžeme také sledovat "zdraví"softwaru v průběhu celého vývoje.

Redukce opakujících se činností je další výhodou. Každá opakující se činnost stojí čas a energii. Typicky je třeba kód vždy zkompilovat, testovat a případně nasadit. K tomu mohou přibýt další stále se opakující činnosti. Zavedením průběžné integrace je velké množství těchto činnosti automatizováno. Díky tomu mohou vývojáři věnovat energii na vytváření a optimalizaci daného projektu.

Zavedení průběžné integrace také posiluje přesvědčení vývojářů. S každým sestavením projektu tým ví, že je jejich práce ověřena testy. Rychle mají přehled, zda vyvíjený projekt správně funguje a zda kód splňuje dané standardy. Díky tomu vidí dopad svého kódu na celý projekt což posiluje motivaci.[10, 3]

2.2 Nevýhody průběžné integrace

Průběžná integrace má mnoho výhod, ale v několika ohledech se může zdát jako zbytečná. Příkladem může být zvýšení režie a údržba integračního systému. Pokud jde o vývoj malého systému, může být zavedení integrace vnímáno jako zbytečná práce navíc. Pokud ale jde o velký, nebo dlouhotrvající projekt určitě se vyplatí investovat do jejího zavedení.

Zavádí-li se integrace do staršího projektu, je potřeba změnit mnoho věcí. Řešením je zavádět integraci pomalu a postupně přidávat tempem, které vyhovuje všem.

Další nevýhodou jsou nároky na hardware a software. Je-li k dispozici hardware, na kterém může integrační server běžet, tak existuje mnoho opensource softwarového řešení. V případě, kdy není hardware k dispozici existují řešení běžící v cloudu. Vždy je na zvážení, jaká investice vyhovuje vývojářskému týmu.

Někdo může říci, že operace běžící na integračním serveru by měli provádět sami vývojáři. Ano měli by, ale je třeba vzít v úvahu, že prováděním těchto činností v odděleném prostředí zvyšuje efektivitu a spolehlivost vyvíjeného softwaru. Navíc zavedením automatických nástrojů se zvýší účinnost a frekvence těchto operací.[3, 2]

2.3 Práce s průběžnou integrací

Aby práce na projektech implementujících průběžnou integraci byla efektivní, musí vývojáři dodržovat jisté pracovní návyky. Existuje několik doporučených praktik jejichž dodržování vyžaduje trochu disciplíny, ale poskytne mnoho výhod.

Často ukládat změny ve zdrojovém kódu. Aby byl zřetelný přínos průběžné integrace je třeba ukládat změny do repositáře často. Čeká-li se víc jak den, tak může docházet k brždění ostatních

členů týmu neposkytnutím posledních změn, které mohou být kritické. Užitečné je dělat pouze malé změny. Tím je myšleno neměnit více komponent najednou, ale pouze jednu a po malých částech. Vzniké změny pak vždy ukládat do repositáře.

Důležitou praktikou je neukládat do repositáře nefunkční kód. Jde o velmi intuitivní myšlenku. Lze ji snadno dosáhnout jednoduchou aktivitou. Vždy před uložením do repositáře kód sestavit na lokálním stroji. Podaří-li se sestavit je pravděpodobné, že nezpůsobí potíže.

Ačkoliv průběžná integrace velmi pomáhá při redukci chyb, mohou se stále objevovat. Nastane-li tato situace je třeba daný problém co nejrychleji odstranit. Právě díky vlastnostem integrace je odstranění problému relativně malý úkon.

Při vývoji systému je třeba psát testy. Důležité je psát tyto typicky jednotkové testy, tak aby mohly být automaticky spouštěny na integračním serveru. Proto je výhodné stanovit si nějaký testovací framework, příkladem NUnit, nebo JUnit framework, pro který se budou testy psát.

V prostředí průběžné integrace má 100% úspěšnost testů stejnou váhu jako úspěšná kompilace projektu. Typicky každý přijme skutečnost, že pokud projekt nelze zkompilovat, tak nemůže fungovat. Totéž platí pro testy v průběžné integraci.

Může nastat situace, kdy se v repositáři objeví nefunkční kód. Pak je vhodné jej nestahovat na lokální stroj. Pokud se tak stane musí vývojář najít a opravit důvod problému což stojí čas, který může být využit jinak. Sice jde o zodpovědnost celého týmu rychle chybu opravit, ale pravděpodobně člověk za chybu zodpovědný na opravě již pracuje. Se spolehlivou komunikaci v týmu lze jednoduše zjistit, zda se na opravě pracuje a vyhnutím se stažení chybného kódu je umožněno dále pracovat na jiných částech kódu.[3, 2, 10]

2.4 Průběžné nasazení

Při vývoji softwaru existuje několik kroků před uvedením do produkce. V průběžné integraci zastřešíme kroky končící spouštěním jednotkových testů. Software pak putuje dál do různých prostředí s různou konfigurací. Příkladem mohou být tato prostředí: kontrola kvality, interní testování, uživatelské testování a samotná produkce softwaru.

Pro každé prostředí existuje jiné nastavení a průběžné nasazení se stará právě o doručování softwaru v platné konfiguraci. Tento automatizovaný proces přispívá k rychlosti vývoje software od tvorby kódu po předáni zákazníkovi.

Často dochází k záměně s průběžným doručením. Tyto dva procesy lze jednoduše rozlišit. Mluvíme-li o nasazení, jde o to, dostat vyvíjený software k produkčnímu týmu, který se pak stará o doručení k zákazníkovi. V případě průběžného doručení jde o proces, kdy prakticky jediným tlačítkem dojde k doručení softwaru přímo k zákazníkovi.[2]

2.5 Extrémní programovaní

Jde o metodologii vývoje softwaru vhodnou pro malé až střední týmy, které se při vývoji musejí potýkat s rychle se měnícími požadavky. Hlavni myšlenkou extrémního programovaní je použití běžně známých principů dotažených do extrému. Příkladem mohou být neustálé revize zdrojového kódu, testovaní jednotek a funkcionality, nebo refaktorizace.[6, 7, 1]

2.5.1 Pilíře extrémního programování

Extrémní programovaní stoji na čtyřech důležitých pilířích. Jsou to komunikace, jednoduchost, zpětná vazba a odvaha.

Komunikace

V rámci projektu je velmi důležité udržet kvalitní komunikaci mezi všemi subjekty. Subjekty pak tvoří vývojový tým, jehož součástí je i zákazník. Nachází-li se zákazník v krátké vzdálenosti od vývojářů a je lehce dosažitelný, pak jde o ideální situaci. V rámci extrémního programovaní jsou použity i techniky, které jsou závislé na dobré komunikaci. Typickým příkladem je párové programovaní.

Jednoduchost

Předpokladem u extrémního programovaní je vyvíjení pravě jen toho, co je opravdu potřeba podle aktuálních požadavků. Nikdy se nevytváří kód typu „v budoucnu se bude nejspíš hodit“. Jednoduchost je velmi obtížná záležitost. Umění nemyslet na předpokládanou budoucnost není velmi jednoduché a lze zde využít princip logické úspornosti tzv. Occamova břitva.

Zpětná vazba

Zpětná vazba je důležitým aspektem v jakékoliv činnosti a to včetně vývoje softwaru. V prvé řadě ji vytvářejí sami programátoři napsáním jednotkových testů. Díky nim lze v relativně krátkém čase získat informace, zda vše funguje jak má a že od poslední změny zdrojového kódu nedošlo k zanesení chyb. Další zpětná vazba pak přichází od zákazníků, kteří používáním softwaru mohou odhalit nedokonalosti, které vývojářům proklouzly.

Odvaha

Odvaha zde nepředstavuje psaní samotného kódu, jako skutečnost, že existuje situace, kdy je třeba zahodit několika denní práci a začít znova. Pro takové akce mohou existovat různé důvody. Například zjištění nevhodnosti použité architektury, špatné rozhraní mezi moduly apod. Pozitivem je, že například po zahozeni celodenní práce dokáže programátor další den pravděpodobně najit optimální řešeni rychleji.

V extrémním programovaní se využívají různé praktiky. V základu jde o 12 praktik, které můžeme rozdělit na tři podskupiny – business, týmové a programovací praktiky.

2.5.2 Business praktiky

Jde o čtyři praktiky zaměřující se především na efektivitu vývoje softwaru. Tyto praktiky označujeme názvy plánovací hra, tým jako celek, malé verze a metafora.

Plánovací hra

Jde o proces rozděleny na dvě části, plánovaní dodávky a plánovaní iterace. Plánovaní dodávky není nic jiného, než odhaleni funkčních požadavků zákazníkem. Každý požadavek je sepsán jako tzv. uživatelský scénář. Po doručení požadavků vývojářům přichází na řadu proces plánovaní iterace. Vývojový tým zpracuje získána data od zákazníka a vytvoří odhad náročnosti a ceny jednotlivých požadavků. Následně po konzultaci se zákazníkem dochází k samotnému návrhu jednotlivých iterací vývoje vzhledem k jejich důležitosti.

Tým jako celek

Jádrem týmu je zákazník, který s týmem na projektu denně pracuje. Například při vývoji softwaru zaměřeného na správu financi by v týmu neměl chybět člověk zabývající se pravě správou financí. Ten pak díky svým zkušenostem představuje přínos pro samotný návrh softwaru.

Malé verze

Extrémní programovaní je založeno na iterativním, přírůstkovém vývoji. Výsledkem iterace je implementované řešení dodané zákazníkovi. Řešení přináší zákazníkovi hodnotu, protože si sám zvolil, které části návrhu budou v iteraci realizovány. Verze jsou dodávány často, a to v rámci dnů, týdnů, nebo měsíců.

Metafora

Tým pracující na projektu si definuje jakousi vizi fungovaní celého systému nazvanou metafora. Ta pomáhá pochopit souvislosti mezi jednotlivými prvky systému. Pro správnou funkci je potřeba aby cely tým znal a používal společnou terminologii a chápal, jak systém pracuje.

2.5.3 Týmové praktiky

Pro práci v týmech je potřeba nastavit jistá pravidla, aby samotná práce vývojáře nepřetěžovala. Unavený vývojář z krátkodobého hlediska není velký problém, nicméně z dlouhodobého hlediska jde o snížení efektivity vývoje. Navíc dochází ke střídání jednotlivých členů, tím pádem je nutné zavést jisté konvence pro psaní kódu.

Párové programování

Na kódu pracují dva programátoři, jeden kód píše a druhý přemýšlí o souvislostech, hledá optimální řešení apod. V průběhu práce na zdrojovém kódu spolu neustále komunikují a také se ve svých rolích střídají. Stejně tak se v rámci práce na celém projektu obměňují i páry. Ač se tato praktika, zda jako neefektivní, tak přináší kvalitnější software, rozšiřovaní znalosti v rámci celého tymu a jakousi robustnost v případě, kdy „odpadne“ člen tymu.

Společné vlastnictví kódu

Díky párovému programovaní může každý člen týmu provést jakoukoliv změnu ve zdrojovém kódu. To znamená dohled většího počtu lidí na každou část kódu. Výsledkem je snížení množství chyb a zvýšení kvality kódu.

Standardy psaní kódu

Pro efektivní fungovaní předchozích dvou praktik je kritické dodržovaní předem definovaných konvencí pro psaní kódu.

Udržitelný vývoj

Extrémní programování prosazuje zdravý vývoj a stojí za názorem, že unaveny programátor vytváří kód obsahující více chyb. Proto je stanovena pracovní doba obecně na 40 hodin týdně. Ovšem počet hodin je silně individuální. Typicky platí jednoduché pravidlo, nemůžete pracovat přesčas, pokud jste přesčas pracovali minuly týden.

2.5.4 Programovací praktiky

Tyto praktiky definují, jakým způsobem se bude nakládat s napsaným kódem. Říkají, jaké zásady při návrhu softwaru dodržovat a jak pracovat s již hotovým kódem.

Průběžná integrace

Programátoři kontrolují svůj kód a několikrát denně ho integruji. Po přidáni kódu do repositáře se spustí testy a ověřuje se správnost. Díky této praktice dochází k rychlému odhaleni chyb.

Jednoduchý návrh

Extrémní programování prosazuje nejednoduší možné řešení. Reálně jde o náročný úkol. Nutnosti tedy je držet se aktuálního plánu iterace a nezahrnovat do vytvářeni kódu budoucí požadavky.

Refaktorizace

V průběhu vývoje se neustále přidávají nové funkce a opravuji se nové a nové chyby. To má za následek postupnou degradaci kódu. Extrémní programování tuto degradaci kódu od-

vrací častým refaktorováním kódu. Tim dochází k neustálému vylepšování a kontrolování kódu. Eliminuji se duplicity, zvyšuje se čitelnost a díky stanoveným standardům psaní kódu není pro členy vývojového týmu problém kód číst.

Vývoj řízený testy

Prvním krokem této praktiky je definice funkcionality a napsáni testů, které ověřují jejich správnost. Testy v první iteraci musí být zákonitě neúspěšné, protože ověřovaná funkcionalita ještě není realizována. Teprve po dokončení všech testů začíná fáze psaní samotného zdrojového kódu. Vývoj řízeny testy požaduje, aby pro každou část kódu byl napsán jednotkový test. Jednotkové testy plní úlohu záchranné sítě při regresním testování. Jsou také podmínkou refaktorizace. Před distribucí musí každý kód úspěšně projít jednotkovými testy.

2.6 Správa verzí zdrojového kódu

V rámci průběžné integrace je verzovací systém stěžejním. Základním úkolem takového systému je zaznamenávaní změn souboru v čase tak, aby bylo možné se vracet k předchozím verzím. Nutno podotknout, že není nutné verzovat pouze zdrojové kódy, ale libovolný typ souboru. Příkladem mohou být celé knihovny, připadně grafické soubory.

Verzovací systémy dělíme na lokální, centralizované a distribuované. Příkladem lokálního systému může být ukládaní jednotlivých verzí do rozumně pojmenovaných adresářů. Takové řešení má vysokou náchylnost k chybám. Uživatel začne zapisovat do špatého adresáře, kopírováním soubory přepíše apod. Pro zamezení takovým chybám byly vyvinuty systémy pro správu verzí založené na jednoduché databázi. Příkladem může být systém RCS založený na ukládání rozdílů mezi podobami souborů.

Při práci v týmech, kdy se kód vyvíjí na více strojích současně, nám lokální systémy nevyhovují. Proto byly vyvinuty centralizované systémy. Používají jeden server pro ukládaní všech verzovaných souboru. Klienti umí z těchto serverů získat data. Koncept centralizovaných systému trpí jedním závažným rizikem. Práce na jediném serveru představuje velkou slabinu v momentě, kdy se cokoliv se serverem stane. Může se tedy stát, že dojde ke ztrátě všech dat. Představitelem centralizovaného řešení je Apache Subversion.

Chceme-li odstranit riziko spojené s jediným serverem můžeme zvolit distribuovaný systém. Oproti centralizovaným systémům uživatel nestahuje jen poslední verzi souboru, ale zrcadlí cely repositář. Při kolapsu serveru pak lze data obnovit od uživatele. Typickým představitelem těchto systému je Git.[4, 5]

2.6.1 Apache Subversion

Jde o opensource systém pro správu zdrojových kódu při vývoji aplikací. Z části je inspirován systémem CVS, je však mnohem flexibilnější a snáz se používá. Je vyvíjen firmou CollabNet. Skládá se ze dvou částí, klientské a serverové. Klientská část slouží pro komunikaci se serverovou části a poskytuje nástroje potřebné pro práci s verzemi v pracovním adresáři. Serverová část se stará o repositář – centrální uložiště. Mezi hlavní přednosti a vlastnosti patří:[11]

- Zachovává většinu funkcí CVS.
- Při každé změně, ať změna obsahu, přesunutí, změna metadat, nebo přejmenování zvýší číslo verze.
- Jako síťový server pro přístup k repositáři může pracovat webový server Apache, kdy pro přístup přes http protokol slouží WebDav/DeltaV protokol.
- Zasílání údajů o rozdílech souborů je prováděno v obou směrech.
- Číslo verze se neudržuje pro každý souboru zvlášť, ale pro celý repositář.

2.6.2 Git

Oproti ostatním verzovacím systémům zpracovává data jinak. Typickým postupem ostatních systému je uchovávat změny pro daný soubor. Oproti tomu Git pracuje na principu "snímku". V podstatě, dojde-li ke změně Git provede "snímek"všech souborů v daném okamžiku a uloží referenci na daný snímek. V případě, kdy nedošlo v souboru ke změně, tak nevytváří nový záznam, ale pouze odkazuje na předchozí identický záznam, který už byl uložen. Git má tedy podobu blížící se souborovému systému oproti verzovacím systémům.

Dalším rozdílem proti ostatním verzovacím systémům je skutečnost, že téměř každá operace je prováděna na lokálních souborech. Práce tedy není závislá na připojení k síti. Důsledkem práce na lokálních souborech je téměř okamžitá reakce při načítání starších verzí. Jednoduše se vyhledá například měsíc starý dokument a Git lokálně vypočte změny. Toto také znamená možnost práce na nových revizích i z míst, kde není připojení, nebo se nelze připojit na firemní VPN. Změny se pak provedou po opětovném připojení. V systémech jako je Subversion, nebo CVS sice lze upravovat soubory off-line, ale nelze tyto změny zapsat do databáze.

V prostředí Gitu nelze měnit obsah jakéhokoliv adresáře nebo souboru, aniž by Git nebyl informován. To je způsobeno skutečností, kdy před každým uložením dochází k provedení kontrolního součtu. K provedení kontrolního součtu Git používá otisk SHA-1 a používá jej jako identifikaci uloženého souboru.

Git typicky data jen přidává. Důsledkem toho je velmi obtížné přimět systém, aby smazal nějaká data, nebo provedl operaci, kterou by nešlo vzít zpět. Tedy zapisují-li se změny pravidelně do databáze a ta se pravidelně zálohuje do jiného repositáře je téměř nemožné přijít o data.

Stěžejní vlastnosti práce Gitu je použití tří základních stavů: zapsáno, změněno a připraveno k zapsáni. Stav "zapsáno"jednoduše říká, že změny jsou bezpečně uloženy v lokální databázi. Jsou-li provedeny změny v souborech, ale nejsou uloženy v databázi jde o stav "změněno". Stav "připraveno k zapsáni"znamená, že soubor s provedenými změnami je určen k uložení do databáze v následující revizi.

Z tohoto rozdělení stavů vyplývá, že projekt uložen v systému git bude rozdělen do tří částí. První a nejdůležitější částí je repositář. Jde o místo, kde Git uchovává metadata a databázi objektů daného projektu. Právě tato část systému je zkopírována při klonováni repositáře do jiného počítače. Další část systému nese název pracovní adresář. Jde o lokální kopii jedné verze projektu. Třetí části je oblast připravených změn. Jde o jednoduchý soubor nesoucí informace o tom, co bude obsahovat následující revize.[5]

standardní postup práce v systému Git vypadá následovně:

1. Změna souborů v pracovním adresáři.
2. Příprava souborů k uložení vložením jejich snímků do oblasti připravených změn.
3. Zápis revizí. Snímky v oblasti připravených změn se trvale uloží do repositáře.

2.7 Testování

Co je testování? Testování je proces získávání informací o vlastnostech a stavu systému za účelem zjištění, zda systém pracuje tak, jak je uvedeno v jeho návrhu, případně identifikaci chyb a jejich předcházení.

V rámci vyvíjeného softwaru potřebujeme dosáhnout určitého stupně jeho spolehlivosti. K ověření spolehlivosti musí nutně docházet k testování. Typickým testem prováděným na integračních serverech je jednotkový test (Unit test). Nicméně je třeba ověřovat i funkčnost jednotlivých komponent a systému jako celku.[1, 3]

2.7.1 Jednotkový test

Jednotkové testy se zaměřují na testování elementární částí kódu např. jedné metody, proměnné, nebo podmínky. Časová náročnost na provádění jednoho testu se pohybuje v rámci vteřin a vývojáři by měl poskytnout detailní zpětnou vazbu o fungování kódu. Jednotlivé testy by měly ověřovat pouze jedinou vlastnost, nebo funkčnost komponenty. Jednotkové testy by měly být plně izolované, tedy nezáleží na pořadí, v jakém jsou spouštěné. Stav prostředí, či služeb ve kterých jsou spouštěné také nesmí ovlivnit jejich výsledek. Další principy, které je vhodné dodržet, jsou jednoduchost a pokrytí. Z kódu testu musí být relativně jednoduše jasné a pochopitelné co ověřuje. Vytváří-li se unit testy, tak by měly pokrývat většinu funkčnosti softwaru. Pro tvorbu jednotkových testů dnes existuje nespočet frameworků. Mezi nejznámější patří JUnit pro Java aplikace a NUnit pro .NET aplikace. V rámci integračního serveru dochází ke spouštění jednotkových testů typicky po každé revizi.[3, 9]

```
[TestMethod]
public void Secti()
{
   Assert.AreEqual(2, kalkulacka.Secti(1, 1));
   Assert.AreEqual(1.42, kalkulacka.Secti(3.14, -1.72), 0.001);
   Assert.AreEqual(2.0 / 3, kalkulacka.Secti(1.0 / 3, 1.0 / 3), 0.001);
}

[TestMethod]
public void Odecti()
{
   Assert.AreEqual(0, kalkulacka.Odecti(1, 1));
   Assert.AreEqual(4.86, kalkulacka.Odecti(3.14, -1.72), 0.001);
   Assert.AreEqual(2.0 / 3, kalkulacka.Odecti(1.0 / 3, -1.0 / 3), 0.001);
}
```

Výpis 1: Příklad unit testu v C#

2.7.2 Test komponent

Testy komponent jsou na rozdíl od jednotkových testů komplexnější. Obvykle trvají déle a zahrnují závislosti například na databázi, souborovém systému, nebo připojení k internetu. Ověření, zda komponenta, sestavená z jedné i více funkcionalit pracuje správně lze považovat za hlavní cíl. Rozdíl oproti jednotkovému testování je skutečnost, že jednotlivé funkcionality komponenty mohou úspěšně obstát v jednotkových testech, ale při testování komponenty jako celku může dojít ke zjištění chybné spolupráce některých částí.

Právě díky nutnosti při testu používat závislá média typu databáze, dochází k vyšší časové náročnosti. Jako příklad poslouží komponenta zastřešující vyhledávání v databázi. Ta testuje různé varianty CRUD operací, kdy pro každou variantu musí dojít k připojení na databázi, zasláni požadavků, obdrženi odpovědi a její následné vyhodnocení. Díky náročnosti testování komponenty jako celku, může čas potřebný pro vykonání testu dosáhnout i jednotek minut. Z tohoto důvodu není příliš vhodné spouštět tento typ testování po každé revizi. Ke spouštění testů komponent typicky dochází v rámci časových cyklů, případně po několika revizích.[3, 2]

```
public class ProjectViewActionTest extends DeftMeinMockStrutsTestCase{
   public void testProjectViewAction() throws Exception{
      this.addRequestParameter("projectId", "100");
      this.setRequestPathInfo("/viewProjectHistory");
      this.actionPerform();
      this.verifyForward("success");
      Project project = (Project)this.getRequest().getAttribute("project");
      assertNotNull(project);
      assertEquals(project.getName(), "DS"); }
   protected String getDBUnitDataSetFileForSetUp(){
      return "dbunit-seed.xml";
   }
   public ProjectViewActionTest(String name){
      super(name);
   }}
```

Výpis 2: Příklad komponent testu s využitím StrustTestCase a DbUnit networku

2.7.3 Systémové testy

Na rozdíl od předchozích jde o nejkomplexnější koncové testy. Jde o testy zaměřené na ověřování funkčnosti celého systému z pohledu koncového uživatele. Typickým příkladem je ověření externích rozhraní typu WWW stránky, koncové služby, nebo GUI systému. Oproti ostatním typům testování nejde o ověření správnosti funkcí jako takových, ale o ověření výsledného systému z pohledu uživatele. Testy jsou náročné na čas a vyžadují použití speciálních nástrojů. Například pro ověření webové stránky lze použít speciální frameworky simulující webový prohlížeč.[3, 2]

```
public class LoginTest extends WebTestCase{
  protected void setUp() throws Exception{
      getTestContext().setBaseUrl("http://pone.acme.com/meinst/");
   }
  public void testLogIn(){
      beginAt("/");
      setFormElement("j_username", "aader");
      setFormElement("j_password", "a1445");
      submit();
      assertTextPresent("Logged in as aader");
   }}
```

Výpis 3: Příklad systémového testu na frameworku JWebUnit

2.8 Automatizace procesů

Jak bylo dříve řečeno v procesu průběžné integrace je primárním úkolem otestovat napsaný kód pomocí testů. Ke spouštění testů potřebujeme mít spustitelný program. Doposud máme pouze kód uložen v repositáři. V následujících kapitolách budou popsaný softwarové řešení, které lze použít. Ve všech případech bude nutné zdrojový kód získat, sestavit a otestovat. Tyto tři záležitosti je třeba automatizovat.

Získání zdrojového kódu

Zdrojový kód se nachází v repositáři. Nastavením pravidel rozhodujeme, kdy dojde k jeho získání. Můžeme chtít, aby se kód získal po každé uložené změně v repositáři. Další možnosti je získávání kódu vždy v pevně určenou hodinu. Například máme-li rozsáhlejší projekt a kompletní testování zabere více času, tak nastavíme pravidla, aby k získání a zpracování došlo vždy o půlnoci, případně v intervalu co 20 hodin.

Automatické sestavování

Sestavování je proces sloužící k přetvoření zdrojového kódu a všech potřebných zdrojů do podoby spustitelného souboru. Můžeme jej rozdělit na dva základní typy:

- **Plné sestavování** - představuje celý proces od začátku, kdy máme k dispozici pouze zdrojový kód. Je tedy nutné získat všechny závislosti a moduly potřebné pro sestavovaný software při každém takovém sestavení.
- **Inkrementální sestavování** - na rozdíl od plného sestavené nedochází vždy k plnému sestavení, ale vychází se z posledního úspěšného sestavení a zpracovány jsou pouze části softwaru ovlivněné změnou. Díky tomuto přístupu je tato metoda mnohem rychlejší.

Dále rozlišujeme události vedoucí ke spuštění samotného procesu. Jde o tři skupiny:

- **Manuální spuštění** - nevhodný pro integrační servery díky nutnosti zásahu uživatele
- **Plánované spuštění** - k procesu sestavení dochází v naplánovaném čase, nebo intervalech. Výhodné u rozsáhlejších projektů, kdy je pro všechny kroky procesu potřeba delšího časového intervalu.
- **Spuštění na základě verzovacího systému** - sestavení se spustí po uložení změn v repositáři. Vhodné u projektů jejichž sestavení a testy nespotřebují moc času, případně nedochází často k revizím.

Pro sestavení zdrojového kódu se typicky využívají programy, které celý proces řídí. Příkladem řídících programů je make, Gradle, Ant, Maven, MSBuild a další.[2, 3]

Automatické testování

Automatické testování spočívá ve spouštění předem definovaných testů nad sestaveným programem získaným z repositáře. Po provedení všech testů nástroj vyhodnotí výsledky, které poskytne uživateli. Mezi výhody tohoto testování patří minimální výskyt lidského faktoru. Ten je omezen pouze na tvorbu testů, případně na jeho vyhodnocení.

Díky tomu dochází k eliminaci chyb a omylů způsobených lidskou nepozorností. Další výhodou jsou nízké provozní náklady na samotný provoz testování. Za hlavní problém automatického testování můžeme považovat vysoké počáteční náklady. To znamená potřebu kvalifikovaných lidských zdrojů na tvorbu testů a s tím spojenou organizační a časovou náročnost celého řešení.[3, 2]

Manuální testování	**Automatické testování**
+ Jednoduchost + Vystačí s málo zkušenou obsluhou	+ Nízké náklady na provoz + Eliminace lidské chyby + Testování velkého počtu vstupů a výstupů
- Časově náročný provoz - Rostoucí náklady s počtem testerů - Nemožnost otestovat vše	- Vysoké počáteční náklady - Nutnost kvalifikovaného personálu - Časová náročnost na tvorbu testů

Tabulka 1: Porovnání manuálního a automatického testování

3 Průběžná integrace – software

Samotná průběžná integrace je pouhá spolupráce jednotlivých částí. Pro jejich spolehlivou práci potřebujeme nějakého správce, který bude vše řídit. Na trhu je velké množství takového typu softwaru. Můžeme však rozlišovat dva základní parametry. Tyto parametry jsou cena a platforma.

Dále bude uvedeno několik nástrojů pro průběžnou integraci. Pro účely demonstrace zprovoznění serveru pro průběžnou integraci jsem zvolil systém Jenkins, kterému bude věnovaná následující kapitola.[4]

CircleCI

Umožňuje vývojářům rychle vypouštět nové verze aplikací, kterým díky testování věří. Díky hostované platformě není třeba vlastnit a spravovat vlastní server.

Platforma:	Hostováno
Cena:	0–750 $ měsíčně
Vznik:	2011
Integrované služby:	Javascript, GitHub, GitHub Enterprise, Node.js, Slack, Python, ...
Uživatelé:	Go Pro, Docker, Facebook, Spotify, Segment, Shopify, ...

Tabulka 2: CircleCI – přehled

TravisCI

Hostovaná, distribuovaná služba určena pro projekty spravované v repositáři GitHub.

Platforma:	Hostováno
Cena:	zdarma pro opensource, jinak 69-489 $ měsíčně
Vznik:	2011
Integrované služby:	GitHub, MySQL, npm, Amazon S3, AWS CodeDeploy, Slack, ...
Uživatelé:	Facebook, Heroku, Mozilla, Twitter, Zendesk, Rails, ...

Tabulka 3: TravisCI – přehled

Bitrise

Platforma jako služba (PaaS) s hlavním zaměřením na vývoj mobilních aplikací (iOS, Android, Xamarin). Každé sestavení spouští vlastní virtuální stroj a po dokončení jsou všechna data smazána, není tedy třeba mít vlastní stroj pro sestavování projektu.

Platforma:	PaaS
Cena:	0-100 $ měsíčně
Vznik:	2015
Integrované služby:	GitHub, Slack, Amazon S3, Ruby, Android SDK, Xamarin, ...
Uživatelé:	Foursquare, Fox, Invision, Product Hunt, The Weather Channel, ...

Tabulka 4: Bitrise – přehled

Visual studio App Center

Platforma:	Hostované
Cena:	zkušební doba zdarma, flexibilní platební plán
Vznik:	2016
Integrované služby:	GitHub, Visual Studio Team Services, Slack, Microsoft Teams, ...
Uživatelé:	Quora, Fox Sports, Highrise, Good Food, FreshDirect, ...

Tabulka 5: VS App Center – přehled

TeamCity

Profesionální, uživatelsky přívětivý integrační server s jednoduchou instalací. Pro malé týmy a open source projekty zdarma. Implementuje možnost spouštět několik sestavení s různými konfiguracemi zároveň.

Platforma:	Web container
Cena:	open source zdarma, jinak od 299$
Vznik:	2006
Integrované služby:	Slack, Amazon Web Services, Docker Cloud, Google Tag Manager, ...
Uživatelé:	eBay, Apple, HP, Airbnb, Salesforce, Stack Overflow, ...

Tabulka 6: TeamCity – přehled

Nevercode

Poměrně mladý nástroj pro průběžnou integraci, který nabízí automatické nastavení pro iOS, Android, Cordovu, Ionic a React Native projekty. Systém dále nabízí šifrováni pro všechny důležitá data.

Platforma:	Web container
Cena:	od 5,99$ měsíčně
Vznik:	2017
Integrované služby:	GitHub, AWS, Slack, Jasmine, iTunes Connect, Gradle Build Tool, ...
Uživatelé:	Mooncascade, Sainsbury's, ingogo, Thunderhead, ...

Tabulka 7: Nevercode – přehled

Bamboo

Systém dovolující vícevrstvé plány sestavování projektů, nastavování spouštěčů v závislosti na revizích. Umožňuje také paralelní spouštění testů.

Platforma:	Web container
Cena:	30 dní zdarma, poté od 10$ ročně
Vznik:	2007
Integrované služby:	AWS CodeDeploy, Docker, Bitbucket, Jira, GitLab, Visual Studio, ...
Uživatelé:	StumbleUpon, Poll Everywhere, Firehose, Interlude, Kaazing, ...

Tabulka 8: Bamboo – přehled

4 Jenkins

Jenkins je jeden z předních open source integračních serveru. Vznikl v roce 2006 pod označením „Hudson“. Typicky běží jako samostatná aplikace ve vlastním procesu s vestavěným Java servletem. Jenkins může být také spouštěn jako servlet v jinem Java servlet kontejneru jako je například Apache Tomcat.

Jenkins tvoří jádro s webovým rozhraním, které je možné rozšiřovat pomocí pluginů. K dispozici je více jak 1400 různých pluginů. V roce 2017 Jenkins překonal 155 000 aktivních instalací, jde však pouze o fragment reálného počtu instalací. Přibližný počet uživatelů dosahuje více než 1,5 miliónu.

Při vytváření úloh není nezbytné mít na vše plugin. Jenkins podporuje dávkové programování i linuxový shell script. Díky této podpoře je možné vytvořit různé funkcionality i bez pluginu

Platforma:	Web container
Cena:	od 5,99$ měsíčně
Vznik:	2006
Integrované služby:	Slach, Testdroid, Hall, Coveralls, AppBlade, Blue Ocean, ...
Uživatelé:	Facebook, Pinterest, HotelTonight, LinkedIn, eBay, Netflix, ...

Tabulka 9: Jenkins – přehled

4.1 Pluginy

Pluginy tvoří jakýsi základní kámen celého systému. Skutečnost, že vývojáři mohou psát vlastní pluginy spolu s velikostí komunity znamená obrovský výběr jíž hotových pluginů. Pluginy je možno procházet na https://plugins.jenkins.io/ včetně jejich popisu. Bohužel instalovat pluginy lze pouze z rozhraní Jenkins.

MSBuild

Plugin povolující použít MSBuild pro sestavení .NET projektů. Po nainstalování je třeba plugin nakonfigurovat zadáním cesty k souboru MSBuild.exe.

Typicky C:\WINDOWS\Microsoft.NET\Framework.

MSTestRunner

Plugin povolující spouštění MSTestu integrovaných do .NET frameworku. V konfiguraci je nutné nastavit cestu k souboru MSTest.exe. Poté lze plugin zařadit jako build step při vytváření úkolu.

MSTest

Plugin sloužící k převodu TRX reportu obdržených po provedení MSTestu. Plugin data analyzuje a převede do čitelné XML podoby.

Gradle

Plugin umožňující spouštět Gradle build scripty. V konfiguraci je možno nastavit ručně cestu, nebo nechat Jenkins automaticky stáhnout a nainstalovat.

GitHub

Plugin integrující Jenkins s projekty uloženými na serveru GitHub. Umožnuje vytvořit propojení Jenkins úlohy s GitHubem, spouštět úkoly po provedeni PUSH na repositáři, případně po úspěšné revizi a ukládat výsledky sestavení zpět na GitHub.

PowerShell

Plugin přidávající podporu pro PowerShell scripty.

Jira Plugin

Plugin přidávající podporu pro plánovací software Jira.

NUnit Plugin

Plugin potřebný pro zobrazení výsledků testů provedených pomoci frameworku NUnit.

LDAP plugin

S tímto pluginem přichází možnost autentizace uživatelů za použití LDAP serverů jako jsou active directory, OpenLDAP a jiné.

Slack Notification

Plugin umožňující zasílat zprávy službě Slack. Výhodné, pracuje-li tým s touto službou. Lze zasílat výsledky testů přímo na channel tomu určený. Pro správnou funkci je třeba provést globální konfiguraci. Je třeba zadat doménu týmu a vygenerovaný Slack token.

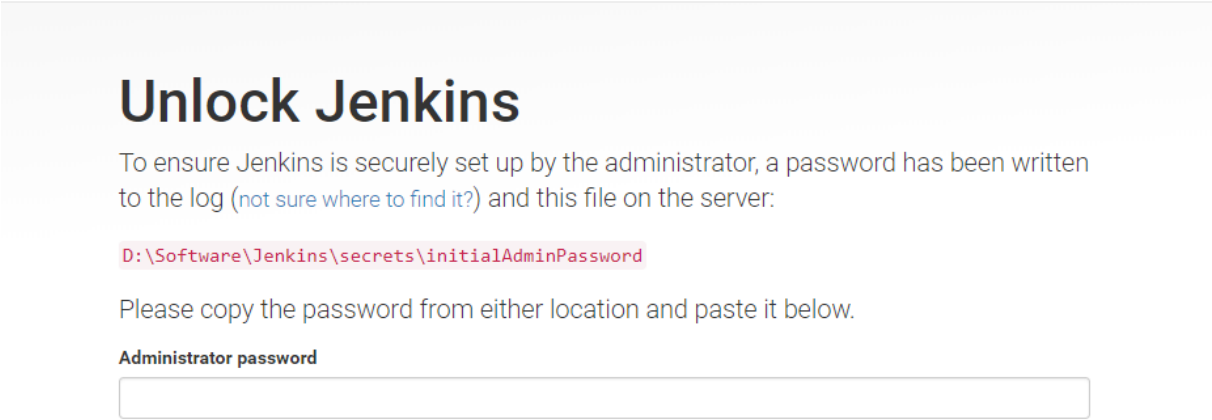
Blue Ocean

Projekt, který kompletně změní design Jenkinse. Primárně se soustředí na vytváření úkolů formou pipeline, ale ponechává podporu pro freestyle úkoly. Vytváření nových úkolu je velmi intuitivní, jednoduché a prakticky na pár kliknutí. V budoucnu by měl plně nahradit aktuální uživatelské prostředí.

4.2 Instalace

Instalace Jenkinse je jednoduchá. Archiv z https://jenkins.io/ se po stažení rozbalí a spustí se instalátor. Po dokončení instalace dojde automaticky ke spuštění Jenkinse a webového prohlížeče na adrese http://localhost:8080/, kde defaultně Jenkins naslouchá.

Při prvním spuštění je bezpečnostních důvodů třeba zadat administrátorské heslo, které se vygeneruje v místě instalace.



Obrázek 2: Jenkins - první spuštění

V následujícím kroku je uživateli předložena možnost stáhnout a nainstalovat komunitou preferované pluginy, nebo si zvolit co se má nainstalovat. Po dokončení instalace všech zvolených pluginu dojde k vytváření uživatelského účtu. Tento krok lze přeskočit, ale typicky se nedoporučuje pracovat s jakýmkoliv systémem pod plnými administrátorskými právy.

Nyní je celá instalace hotova a Jenkins je připraven pro použití. Nicméně připravený Jenkins neznamená, že budou plně fungovat pluginy. U většiny musíme provést konfiguraci, zadat cesty k souborům, přidat přihlašovací údaje a podobně. Tyto konfigurace se provádějí v menu Manage Jenkins -> Configure System, nebo Manage Jenkins -> Global Tool Configuration s výjimkou nastavení jazyka prostředí. Toto nastavení je automatické a řídí se výchozím jazykem používaného prohlížeče.

4.3 Příklad vytvoření úlohy

V následujících kapitolách bude představen postup, jak vytvořit úlohu na integračním serveru Jenkins. Půjde o ukázku freestyle projektu, náročnějšího na nastavení a pipeline projektu, typu úlohy, pro kterou je uzpůsobené rozhraní Blue Ocean. Pipeline projekt nese několik výhod oproti freestyle projektu. Není třeba otrocky vše nastavovat na integračním serveru, ale stačí vytvořit konfigurační soubor "Jenkinsfile"a uložit jej v repositáři.

Jenkinsfile je textový soubor obsahující Groovy script řídící celou úlohu. Další výhodou pipeline projektu je rozdělení celého integračního procesu do etap, které mohou běžet i paralelně. Můžeme například spouštět testy na více větvích zdrojového kódu. Dojde-li k chybě v nějaké etapě, tak hned vidíme ve které, což usnadňuje případné hledání chyb.

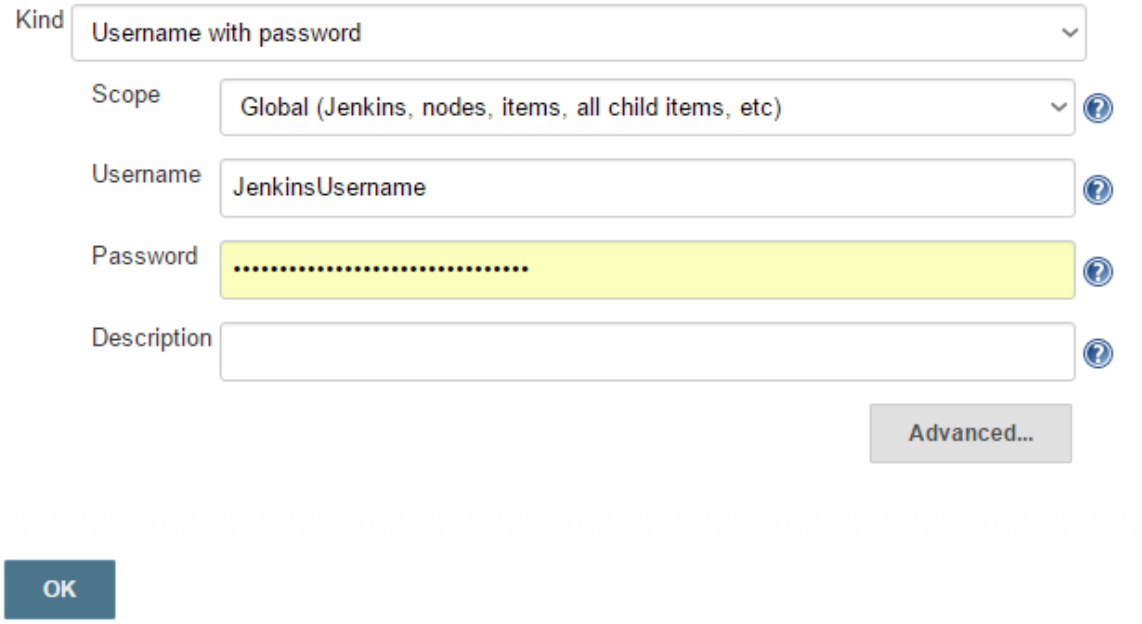
Chceme-li vytvořit úlohu, která bude každý den o půlnoci stahovat zdrojový kód .NET projektu z GitHubu a spouštět unit testy uložené v rámci projektu, budeme potřebovat několik věcí. Předně nainstalovaný Jenkins a nástroje potřebné pro sestavení .NET projektu. V neposlední řadě také .NET framework a nástroje pro spuštění testů.

Jenkins typicky běží na serveru. Musíme na něj tedy nainstalovat .NET Framework používané verze, dále Agents for Microsoft Visual Studio 2015 pro získání MSTest bez nutnosti Visual Studia. Alternativou k MSTestu může být využití testovacího frameworku NUnit. V následujících příkladech bude využito obojí. Nainstalujeme tedy i NUnit framework. Nakonec Microsoft Build Tools, které nám obstarají sestavování projektu. Na server bude také nutno umístit nástroje nuget a git. Pro jednodušší práci s nástroji git a nuget je vhodné uvést jejich lokaci do systémové proměnné Path.

Ze softwaru je to vše, dále je třeba nainstalovat a nakonfigurovat potřebné pluginy do Jenkinse. Jde o pluginy GitHub Integration, MSBuild Plugin a MSTest Plugin.

4.3.1 Git Server Credentials

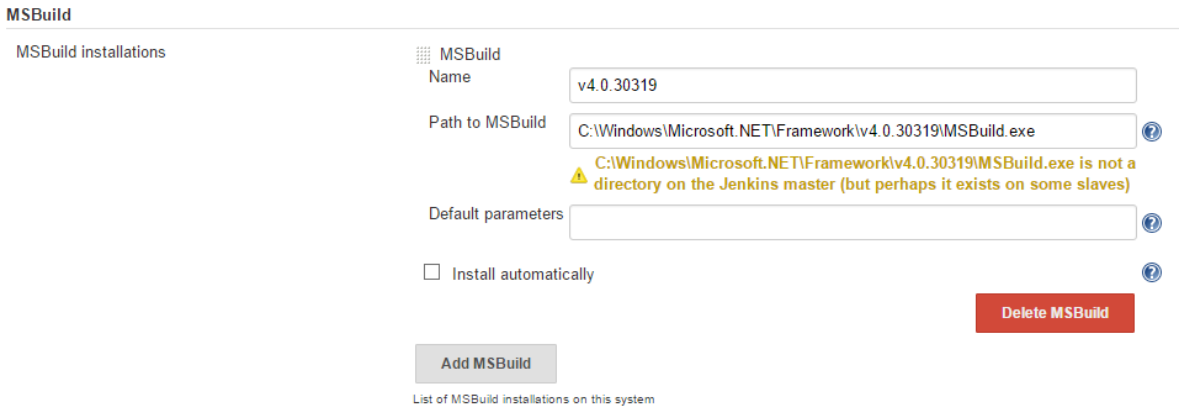
Pro přístup Jenkinse na GitHub je třeba vytvořit pověření. V menu Manage Jenkins -> Manage Credentials -> Add Credentials vyplníme uživatelské jméno a heslo na GitHub. Tento účet následně použijeme při nastavování Git pluginu.



Obrázek 3: Jenkins - přidání přihlašovacích údajů na GitHub

4.3.2 Konfigurace MSBuild

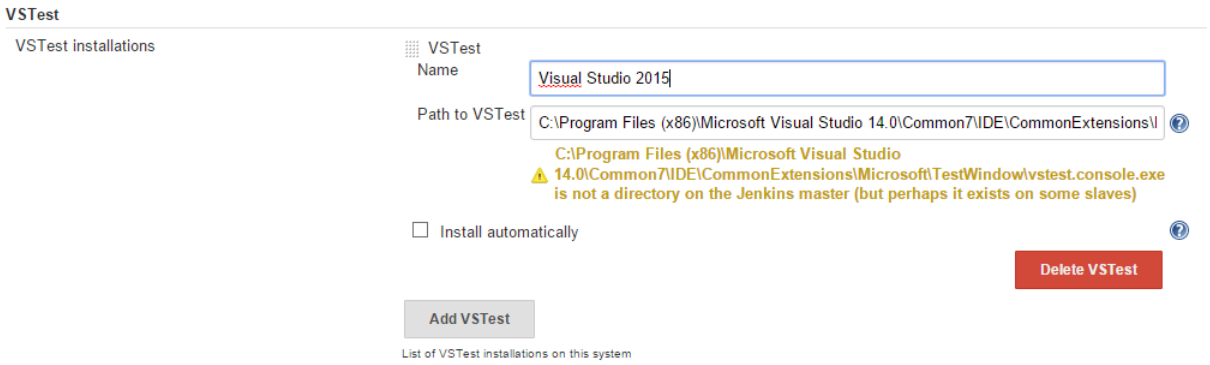
Přejdeme do menu Manage Jenkins -> Global Tool Configuration. Zde v části MSBuild přidáme MSBuild a nastavíme název, odpovídající verzi (pro lepší orientaci) a cestu k souboru MSBuild.exe



Obrázek 4: Jenkins - nastavení MSBuild pluginu

4.3.3 Konfigurace VSTest

Na stejném místě jako MSBuild najdeme část VSTest a přidáme podobně jako MSBuild.

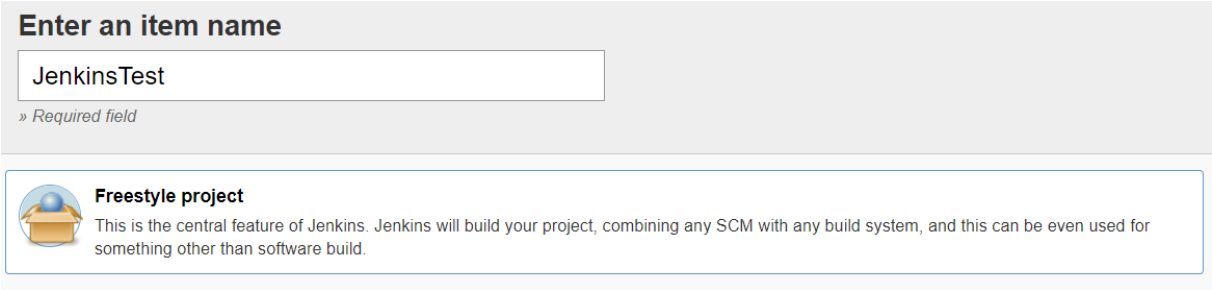


Obrázek 5: Jenkins - nastavení VSTest pluginu

4.3.4 Vytvoření úlohy – Freestyle projekt

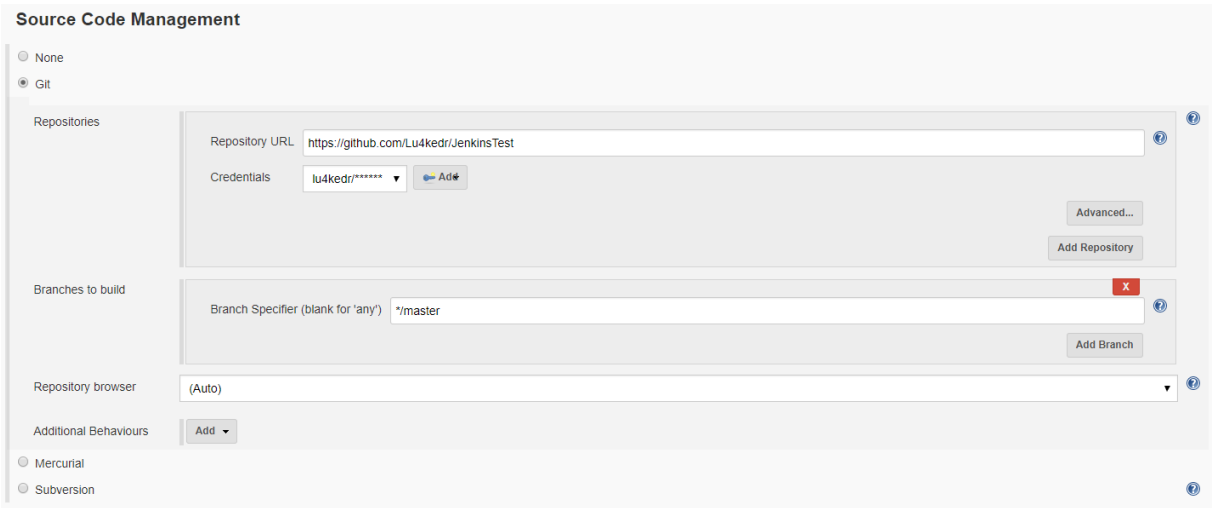
Nyní, když máme cesty k potřebným nástrojům nastavené, můžeme přejít k nastavené samotné úlohy. jedná se o freestyle projekt, který po nastavení stáhne z GitHubu aktuální verzi zdrojového kódu, pomocí nuget nástroje stáhne všechny reference následně projekt sestaví a spustí unit testování. Dojde-li k selhání zašle email s výsledky úlohy.

V menu Jenkinse zvolíme New Item, zadáme název, typ úlohy (Freestyle projekt) a potvrdíme.



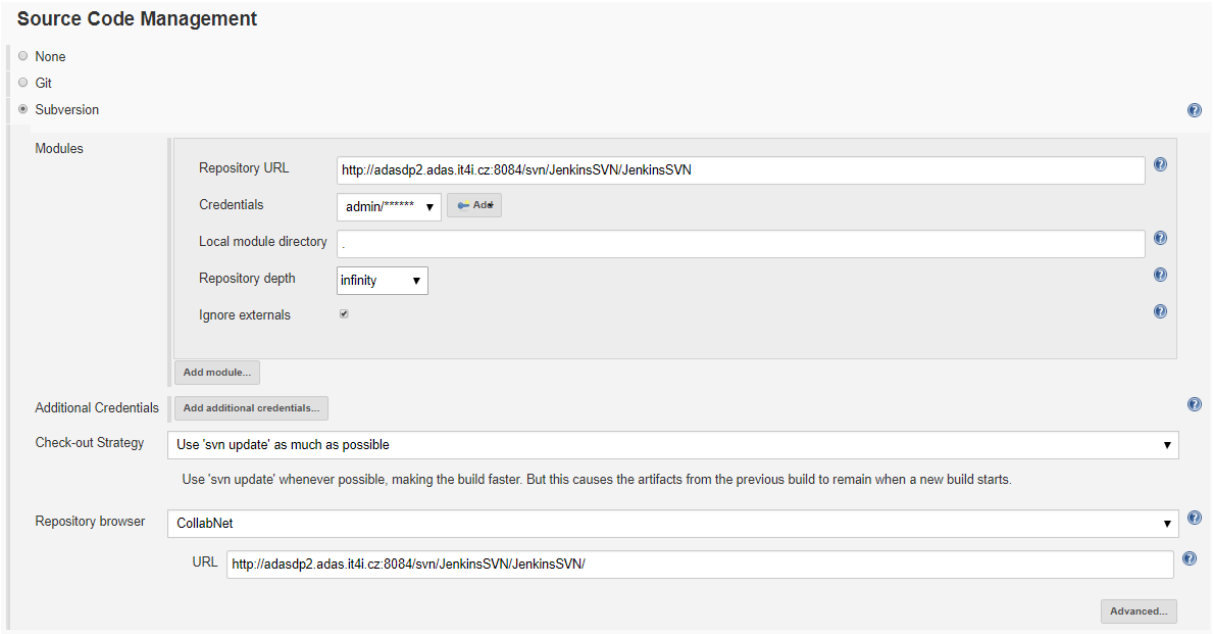
Obrázek 6: Jenkins - nová úloha (freestyle projekt)

Následuje samotná konfigurace úlohy. V první části zvolíme pouze GitHub projekt a zadáme adresu projektu na www.github.com. Dále v části "Source Code Managment"zvolíme Git, zadáme url adresu repositáře a vybereme dříve vytvořený účet.



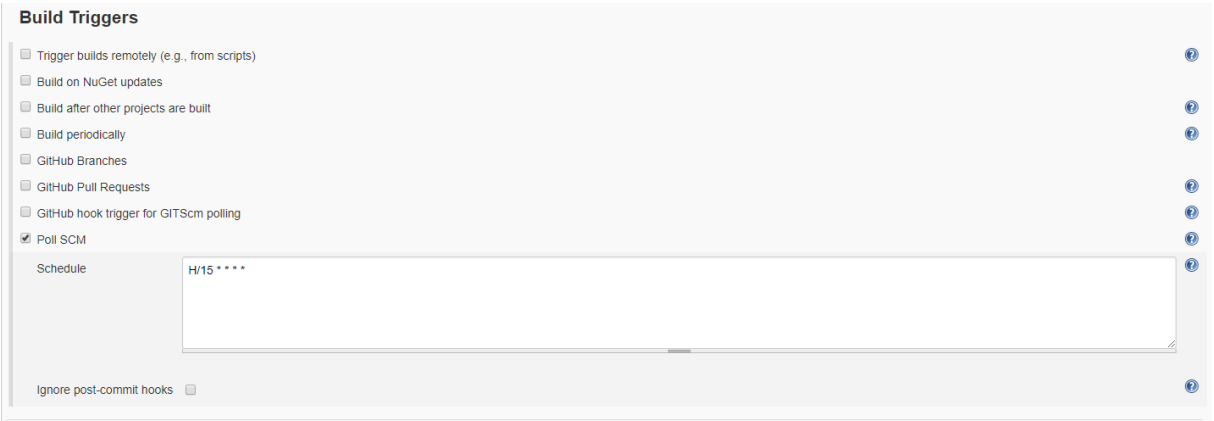
Obrázek 7: Jenkins - Správa zdrojového kódu(GitHub)

V případě, kdy pro správu zdrojového kódu využíváme systém Subversion, zvolíme možnost Subversion, vyplníme URL a zadáme přihlašovací údaje. Výsledné nastavení může vypadat nějak takto:



Obrázek 8: Jenkins - Správa zdrojového kódu(Subversion)

Další část s názvem "Build Triggers"nám umožnuje nastavit, co bude výslednou úlohu spouštět. My budeme chtít spouštění co 15 minut. Zvolíme tedy "Pool SCM"a zadáme kdy chceme úlohu provádět. Detailnější informace o formátu zápisu jsou v nápovědě vedle textového pole. Pro spouštění každých 15 minut zadáme "H/15 * * * * "

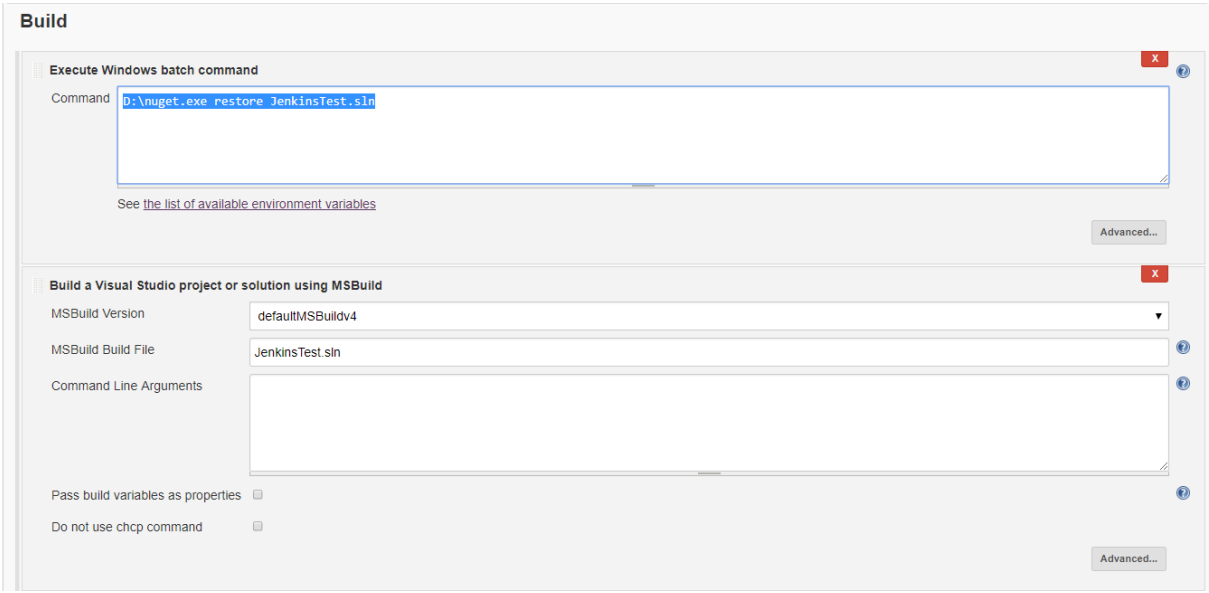


Obrázek 9: Jenkins - spouštěče úlohy

Nyní se dostáváme k nastavení samotného sestavování projektu, část "Build". V první řadě, jedná-li se o .Net projekt, musíme získat všechny potřebné reference. K tomu nám pomůže správce balíku Nuget.

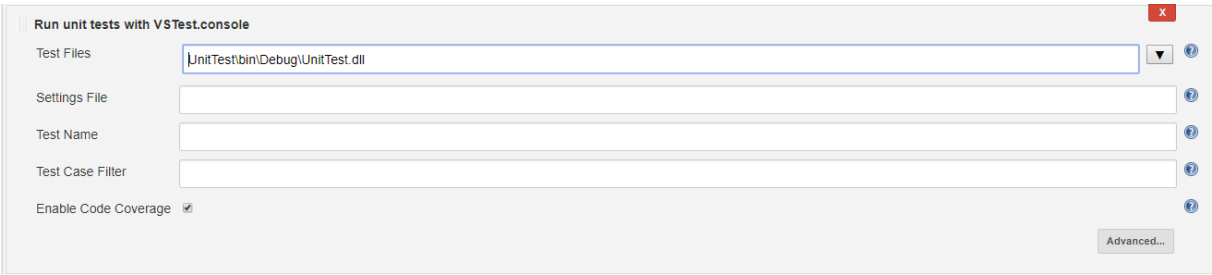
Přidáme tedy nový "build step"a zvolíme "Execute windows batch command". Do command pole musíme uvést cestu k souboru nuget.exe, parametr restore a název souboru *.sln daného projektu. Například "D:\nuget.exe restore JenkinsTest.sln".

Tímto máme zajištěno, že samotné sestavení projektu neskončí vinou chybějících referencí. Přidáme tedy další krok, ale tentokrát vybereme "Build Visual Studio project or solution using MSBuild". Vybereme verzi, kterou jsme dříve konfigurovali a zadáme název solution souboru projektu, který se bude nacházet v kořenové složce úlohy



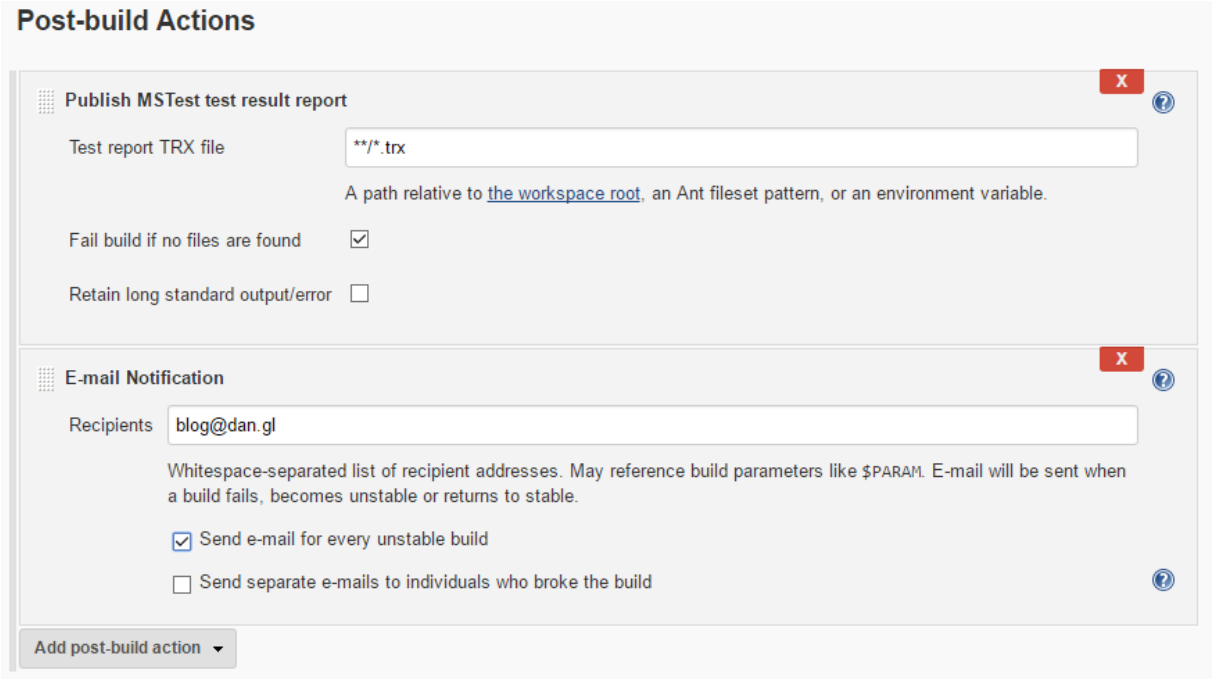
Obrázek 10: Jenkins - nastaveni Build sekce

Nyní máme stáhnutý a sestavený projekt. Nezbývá tedy než spustit testy. Chceme testy spouštět skrze VSTest. Přidáme další krok s názvem "Run unit tests with VSTest.console". Zvolíme verzi, jež jsme přidali v konfiguraci pluginu a zadáme cestu k testovacím souborům. Ty byly vytvořeny při sestavení ve složce Debug. Přesná cesta se odvíjí od zvolených názvů v projektu. V případě, že bude špatně uvedena, skončí úloha chybou. Nicméně stačí na dané úloze otevřít workspace a podívat se na přesnou cestu.



Obrázek 11: Jenkins - nastaveni VSTest

Posledním krokem před konečným uložením úlohy je přidáni "Post-build"kroku. Zde přidáme "Publish MSTest test result report"a následně "Email Notification". Vyplníme maily, kam se má zaslat zpráva o výsledcích, zvolíme volbu "Send email for every unstable build"a nakonec uložíme.



Obrázek 12: Jenkins - nastaveni reportu

Pro ověření zvolíme "Build Now"a v části menu "Build history"se objeví nová položka. Když ji otevřeme a zvolíme "Console Output"uvidíme celý proces, který jsme nastavili. V případě, že všechny testy budou úspěšně dokončeny, označí se úloha modře. Dojde-li k chybám označí se červeně a zašle se report emailem.

V Jenkinsu pak máme přehled o jednotlivých úlohách včetně vyobrazení úspěšných a neúspěšných instancí. Na následujícím obrázku [13] vidíme červené a modré indikace posledních pokusů, sloupec S-status. Ve vedlejším sloupci W-weather (počasí) je vyobrazen celkový stav úlohy. "Počasí"se pak odvíjí od poměru úspěšných a neúspěšných instancí. Úlohy "cppTest"a "JenkinsSVN"jsou dlouhodobě neúspěšné. Obě úlohy vychází ze stejného projektu, kde je úmyslně zavedena chyba v unit testování z experimentálních důvodů. Jediným rozdílem v těchto dvou úlohách je použití správy zdrojového kódu. Úloha "cppTest"je uložena na GitHubu, kdežto "JenkinsSVN"používá Subversion.

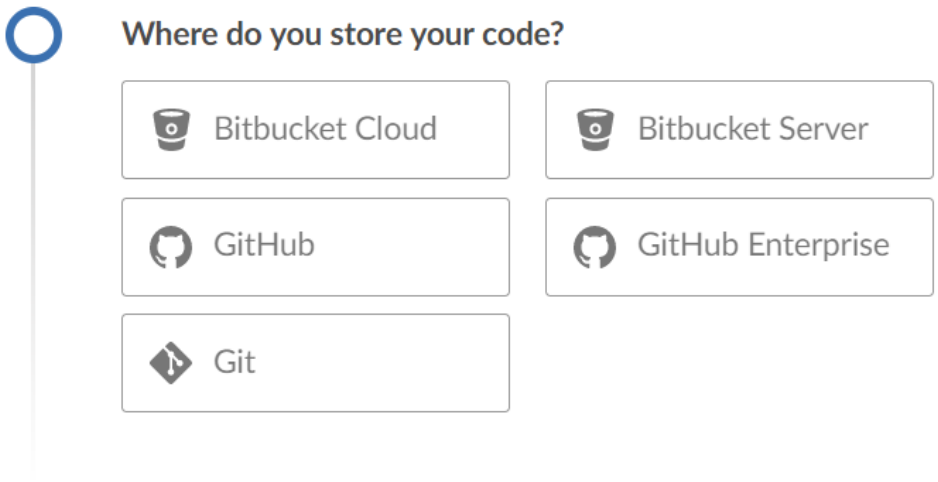
All

S	W	Name ↓	Last Success	Last Failure	Last Duration
		cppTest	4 mo 23 days - #7	4 mo 23 days - #15	11 sec
		JenkinsSVN	4 mo 23 days - #23	17 hr - #168	30 sec
		Test	5 mo 14 days - #35	5 mo 14 days - #33	19 sec
		test3	3 days 14 hr - #3	N/A	2.2 sec
		Test_2	5 mo 13 days - #3	5 mo 5 days - #4	25 sec

Obrázek 13: Jenkins - přehled úloh a jejich stavy

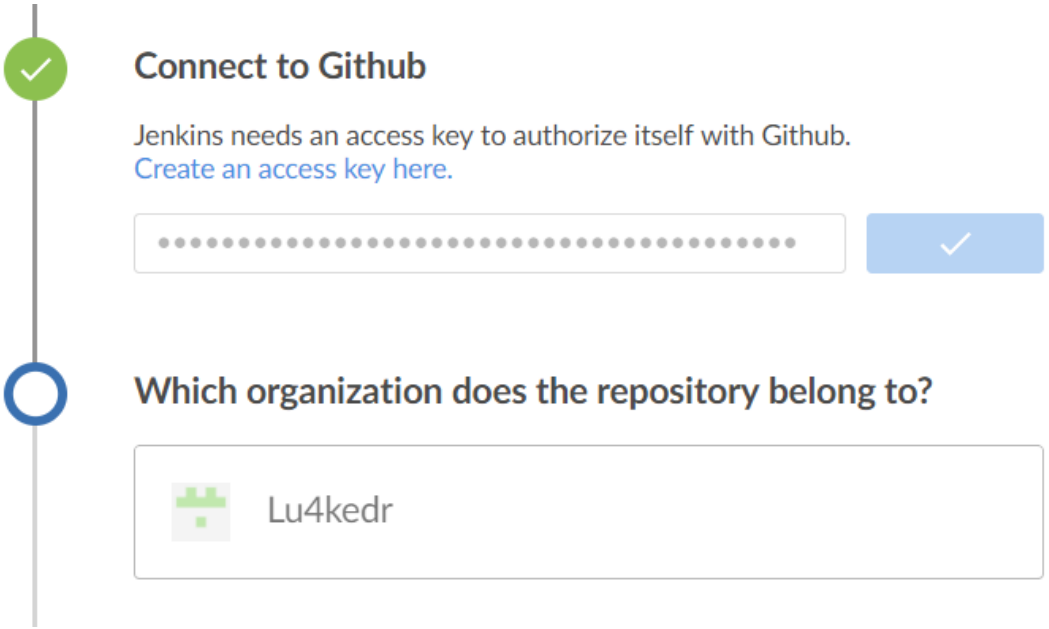
4.3.5 Vytvoření úlohy – Pipeline projekt

Pro vytvoření pipeline projektu můžeme využít i klasické uživatelské rozhraní Jenkinse. Nicméně máme nainstalován plugin Blue Ocean, tak jej využijeme. Přepneme tedy do rozhraní Blue Ocean a vytvoříme novou pipeline. Projekt máme uložen na GitHubu, zvolíme tedy GitHub. Následně



Obrázek 14: Vytvoření pipeline

budeme vyzváni k zadáni "GitHub access token". Token si necháme vygenerovat v Nastavení profilu na GitHubu -> Developer settings -> Personal access tokens. Po získáni tokenu jej zadáme a vybereme repositář se kterým chceme pracovat. V následujícím kroku Jenkins hledá



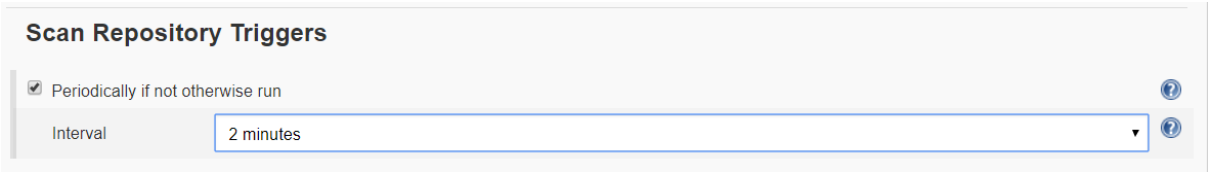
Obrázek 15: Jenkins(Blue Ocean) - vytvoření pipeline

v repositáři soubor Jenkinsfile. Pokud soubor není nalezen dojde ke spuštění tvorby nového. Nicméně přidáni Jenkinsfile do repositáře se jeví mnohem efektivněji. Samotný script pak může vypadat následovně:

```
pipeline{
   agent any
  stages{
      stage('Nuget Restore'){
         steps         {
            bat 'Nuget restore Jenkins_PiplineTest.sln'
      }
    }
    stage('MSBuild'){
      steps         {
        bat "\"${tool 'MSBuild'}\" Jenkins_PiplineTest.sln /p:Configuration=
            Debug /p:Platform=\"Any CPU\" /p:ProductVersion=1.0.0.${env.
            BUILD_NUMBER}"
      }
    }
    stage('Test'){
      steps{
        bat 'nunit3-console.exe NUnit.Tests/bin/Debug/NUnit.Tests.dll'
      }
    }
      stage('Results'){
      steps{
        nunit(testResultsPattern: 'TestResult.xml')
      }
    } }
  post{
      success{
         echo 'Successful'
      }
    failure{
        mail bcc: '', body: 'Build FAILED', cc: '', from: 'Jenkins', replyTo: '
            ', subject: 'ERROR CI', to: 'dra0060@vsb.cz'
    }
   } }
```

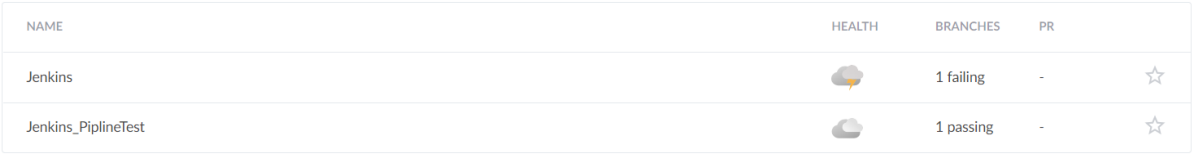
Výpis 4: Jenkinsfile

Je-li potřeba úlohu spouštět ve specifický čas, nebo interval, musíme do scriptu přidat část vymezující trigger podmínky. Pokud postačuje spouštění po změně v repositáři, tak stačí přejít do nastavení úlohy, povolit periodické hlídáni repositáře a nastavit interval.

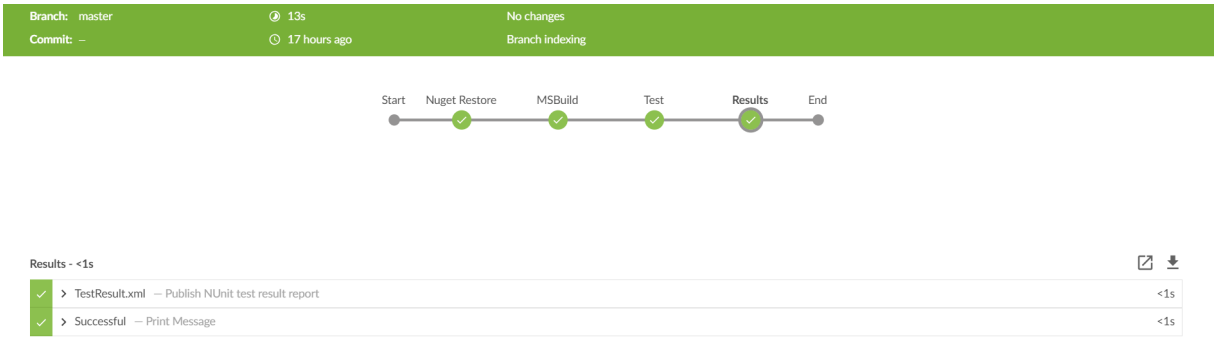


Obrázek 16: Jenkins(Blue Ocean) - hlídání repositáře

Výsledky jednotlivých úloh můžeme zobrazit stejně jako v předchozí úloze. Nicméně pro vytváření úlohy bylo použito rozhraní Blue Ocean, kde jsou výsledky přehledně zobrazeny a v případě selháni lze okamžitě poznat, ve které části k problému došlo.

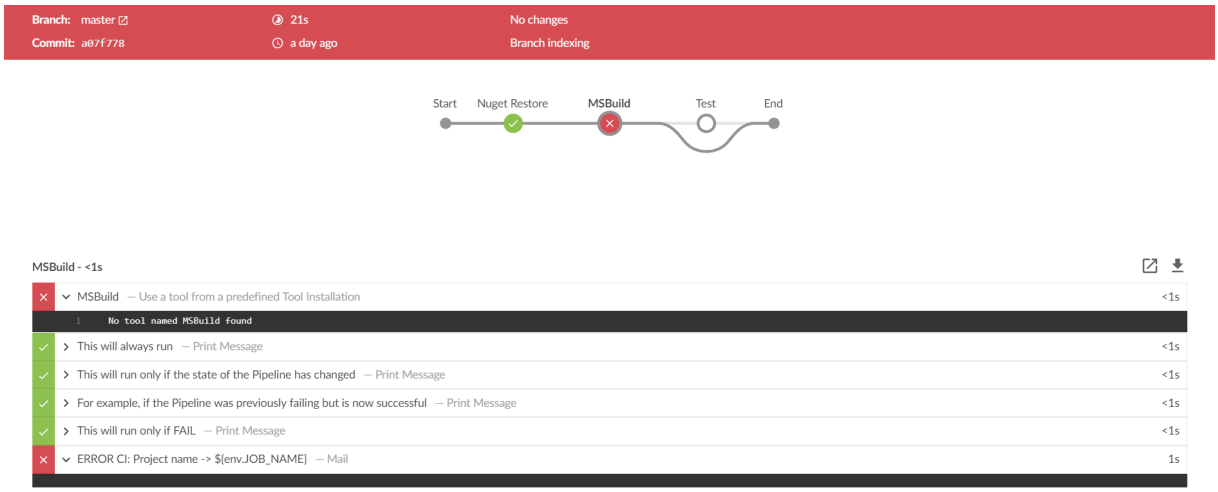


Obrázek 17: Jenkins(Blue Ocean) - přehled úloh



Obrázek 18: Jenkins(Blue Ocean) - úspěšné vykonání úlohy

Na následujícím obrázku [19] je znázorněn průběh úlohy, kdy došlo k chybě způsobené ne nakonfigurováním nástroje MSBuild.



Obrázek 19: Jenkins(Blue Ocean) - chybějící MSBuild

5 Závěr

Průběžná integrace se na první pohled zdá být jednoduchou problematikou. Ovšem podíváme-li se na všechny aspekty, jde o komplexní metodiku ověřenou praxi. Skutečnost, že na trhu je velké množství produktů zaměřujících se právě na realizaci průběžné integrace napovídá, že jde žádanou oblast vývoje software.

Tato práce ukazuje, jak komplexní vývojová disciplína je průběžná integrace. Podává přehled o aktuálně používaných nástrojích v rámci použití průběžné integrace.

Mimo teoretické informace tato práce demonstruje základní nastavení vybraného softwaru Jenkins pracujícího jako integrační server. Součásti práce bylo zároveň konfigurování a otestování integračního serveru ve spolupráci s verzovacími systémy GIT a SVN. Spolupráce s oběma systémy byla úspěšná jak pro úlohu freestyle projekt, tak pro pipeline projekt.

Literatura

[1] MARTIN, Robert C. a Micah. MARTIN. Agile principles, patterns, and practices in C#. Upper Saddle River, NJ: Prentice Hall, c2007. ISBN 0-13-185725-8.

[2] HUMBLE, Jez. a David FARLEY. Continuous delivery: reliable software releases through build, test, and deployment automation. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 978-0-321-60191-9.

[3] DUVALL, Paul M., Steve MATYAS a Andrew GLOVER. Continuous integration: improving software quality and reducing risk. Upper Saddle River, NJ: Addison-Wesley, c2007. ISBN 978-0-321-33638-5.

[4] SINK, Eric. Version control by example. Champaign, Ill: Pyrenean Gold Press, 2011. ISBN 9780983507918.

[5] CHACON, Scott a Ben STRAUB. Pro Git: Everything you need to know about git [online]. 2. 2018 [cit. 2018-04-23]. Dostupné z: https://git-scm.com/book/en/v2.

[6] BUCHALCEVOVÁ, Alena. Metodiky budování informačních systémů. Praha: Oeconomica, 2009. ISBN isbn9788024515403.

[7] BECK, Kent. Extrémní programování. Praha: Grada, 2002. ISBN isbn9788024703008.

[8] DUCHO, Pavel a Tomáš ZAJÍČEK. Výhody kontinuální integrace a automatizovaného deploymentu z pohledu webové integrace. Webová integrace [online]. 2015 [cit. 2018-04-23]. Dostupné z: http://www.web-integration.info/cs/blog/vyhody-kontinualni-integrace-a-automatizovaneho-deploymentu-z-pohledu-webove-integrace/

[9] PEREZ, Raul. Unit testing, component level testing and UI testing, what to use and when [online]. 2010, , 3 [cit. 2018-04-23]. Dostupné z: https://blogs.msdn.microsoft.com/raulperez/2010/04/29/unit-testing-component-level-testing-and-ui-testing-what-to-use-and-when/

[10] TAYLOR, Andrew. 5 Advantages of Continuous Integration [online]. 22.4.2017 [cit. 2018-04-23]. Dostupné z: https://pantheon.io/blog/5-advantages-continuous-integration

[11] PILATO, C. Michael. Version control with subversion. 2nd ed. Beijing: O'Reilly, 2008. ISBN 978-0-596-51033-6.